Partition Testing Does Not Inspire Confidence

Dick Hamlet, *Member, IEEE,* and Ross Taylor

Abstract—**Partition testing, in which a program's input domain is divided according to some rule and tests conducted within the subdomains, enjoys a good reputation. However, comparison between testing that observes subdomain boundaries and random sampling that ignores the partition gives the counterintuitive result that partitioning is of little value. In this paper we improve the negative results published about partition testing, and try to reconcile them with its intuitive value. Theoretical models allow us to study partition testing in the abstract, and to describe the circumstances under which it should perform well at failure detection. Partition testing is shown to be more valuable when the partitions are narrowly based on expected failures and there is a good chance that failures occur. For gaining confidence from successful tests, partition testing as usually practiced has little value.**

Index Terms—**Partition testing, random testing, reliability, software testing theory.**

I. Partition Testing

INPUT partitioning is the natural solution to the two fundamental testing problems of systematic method and test volume. By dividing a program's input domain into classes whose points are somehow "the same," it is sufficient to try one representative from each class; the problem of systematic testing is reduced to a proper definition of the classes. A partition can be defined using all the information about a program. It can be based on requirements or specifications (one form of "blackbox" testing), on features of the code ("structural" testing), even on the process by which the software was developed, or on the suspicions and fears of a programmer. For example, a specification-based partition might divide the input domain into inputs required to invoke one of several software features $F_1, F_2, \cdots$. Or, a binary structural partition might consider inputs that do or do not force use of a suspect data structure. The partition-testing method forms the intersection of such input classes—e.g., one class formed from the above would be those inputs requiring feature F_2 *and* making use of the suspect data structure; another would be inputs for feature F_1 and not making use of it. The goal is to make the resulting classes so narrow that each aspect of the program, of the specification, of development, each programmer concern, etc., is separated into a unique class.

Goodenough and Gerhart [1] expressed a partition using "significant predicates" from both specification and program to divide inputs into classes, and created the intersection by considering all combinations of predicate values. Weyuker and Ostrand [6], and Richardson and Clarke [2] describe the special case of intersecting specification classes with those defined by program path predicates. Although "partition testing" often carries a connotation of functional testing involving specifications, here we use the term in the general sense of any input-space division. The technical results require that this division be a partition in the mathematical sense; that is, a division of the input domain into disjoint, mutually exhaustive classes. (Technically, the partition must be an equivalence relation, which has those equivalence classes.) Some kinds of "functional testing" based on specifications, and path testing, meet the technical requirements.

In the sequel, we reserve the word "partition" for an equivalence relation, and we call its equivalence classes "subdomains" as well as "classes." But because of long standing usage, we will continue to use "partition testing" to refer to any testing method that subdivides the input domain and chooses test points from each subdomain. In Section II-B we consider testing schemes like statement coverage and mutation that are not partitions, and in Section IV-B we try to convince the reader that it is not far wrong to apply our technical results to any intuitive form of partition testing.

The strength of partition testing is its ability to use any and all information, and to examine information in combinations that may not have been thought of during development. Intuitively, the source of program bugs is some unlikely combination of requirements, design, and programmer inattention. By including these factors in the subdomain definition, it seems that nothing has been missed in testing. Good subdomains are defined and refined throughout development as information arises.

Partition testing can be no better than the information that defines its subdomains. For the method to work perfectly, all inputs in one subdomain must be interchangeable—if one causes a failure, any other must do the same. (Goodenough and Gerhart called this property "reliability," but since that word has other definitions, let us here call a subdomain *homogeneous* if either all its members cause the program to succeed or all cause it to fail. Weyuker and Ostrand used the term "revealing" for a homogeneous subdomain.) Intuitively, homogeneity might be obtained by narrowing the subdomain definitions. It seems that enough constraints would force test points from a subdomain to behave the same. In practice, when partition testing goes wrong, it is technically because a subdomain lacked homogeneity—the test point succeeded but another point in the subdomain fails in the field. The subdomain definition can always be blamed for the debacle: some refinement would have separated the success and failure points.

Practical partition testing, with less-than-perfect subdomains, must thus sample each subdomain often enough to improve the chance of detecting failures. A uniform distribution across each subdomain seems appropriate, because each subdomain represents an indivisible collection. Were the appropriate distribution skewed in any way it would be a basis for further refinement. The criteria by which subdomains are defined are a mixture of seeking to establish confidence in the program (Goodenough and Gerhart had this orientation), and to find its failures (as Weyuker and Ostrand proposed [6] and as Myers [9] strongly advises). On the one hand, specification-based blackbox criteria and many structural criteria strive for "coverage"—these

Manuscript received November 13, 1989; revised July 24, 1990. Recommended by M. S. Deutsch. The work of D. Hamlet was supported by the National Science Foundation under Grant CCR-8822869.

D. Hamlet is with the Department of Computer Science, Portland State University, P.O. Box 751, Portland, OR 97207.

R. Taylor is with Tektronix, Inc., P.O. Box 1000, M/S 63/356, Wilsonville, OR 97070.

IEEE Log Number 9039335.

are confidence methods. But subdomains that concentrate on a subroutine expected to be fault-prone, or on a difficult aspect of a data structure (e.g., collisions in a hash table), are seeking failures.

The distinction between seeking confidence and seeking failure is central to evaluation of testing methods. It turns on the test inputs used, their distribution over the input domain. When looking for failures, it is appropriate to use peculiar test inputs. So-called "special-values testing" is just a name for choosing strange test points likely to provoke behavior problems. However, tests seeking to excite failure are not representative of a program's day-to-day operation. Confidence in daily performance can be gained only by testing that mimics the "operational distribution" of typical usage. Thus the two kinds of testing may be expected to have little in common: tests designed to expose failures are not representative, and tests representative of typical operation may seldom encounter failure situations. The technical results presented below deal with this important distinction. Briefly, we show that partition testing is not much different than random testing; hence success of a few partition tests, however clever, does not contribute to confidence in the program. The other side of the coin is that unless care is taken, partition testing may not even succeed in exposing failures. Using random testing as a standard, we suggest how to improve the failure-finding ability of partition testing.

This study was undertaken because partition testing did not live up to its intuitive value in two earlier studies. In their brief for random testing [3], Duran and Ntafos published a precise comparison between it and partition testing. Their surprising result is that the two methods are of almost equal value, under assumptions that seem to favor partition testing. Random testing has a decidedly spotty reputation, probably because it makes almost no use of special information about the program being tested. It is certainly counterintuitive that the best systematic method is little improvement over the worst. Hamlet [5] corroborates this result using a different sampling model. He shows random testing to be *superior* to partition testing, its superiority *increasing* with more partitions and with the program confidence required.

These results are analyzed and extended in Section III; close examination strengthens them and gives insight into what makes partition testing work. In brief, quantitative results about partition testing are counterintuitive because our intuition is untrained in confidence testing. To guarantee high confidence in even medium-scale software requires very large test sets to be executed without any failures, and for this situation our intuition fails. Partition testing's systematic nature may also be overrated. The intuition that the "right" partition will find most of the faults may be no more than wishful thinking. Any program has a finite number of faults, and hence a partition exists to expose them all. But if the right subdomains are the ones that find the unknown faults, those subdomains are equally unknown, and no system can necessarily find them.

II. Properties of Partition Testing

In this section we explore some underlying strengths and weaknesses of partition testing.

A. "Treated the Same" Subdivisions

Partition testing works perfectly when the right points are chosen from the right subdomains. But a practical partition must be obtained *before* a program's bugs are identified. Analysis that shows a partition to exist for detecting a known fault falls under the archery scheme proposed by Walt Kelly: shoot first and paint the target where the arrow falls. Partitions can be *defined* to necessarily reveal faults (if they exist); this is the view of fault-based testing suggested by Weyuker and Ostrand [6], and investigated by Morell [7]. Fault-based testing is the best candidate for a formal method that combines proving and testing, but its theory is more like proving than testing. As we will now show, perfect partitions seem to share this quality.

In the sequel we use the convention that programs have a single input value and a single output, so that their meanings are functions of one variable. We use the Mills notation [8]: $\boxed{P}$ denotes the function that program P means. Specifications are also assumed to be functions, so that a program P is correct with respect to specification S iff $S = \boxed{P}$. These assumptions simplify the presentation; the results do not change in the more realistic case of multiple input/outputs and relational specifications.

We can learn a good deal by considering the simplest case of a partition with homogeneous classes in which inputs are literally treated the same. In the simplest program view, inputs are treated the same when they yield the same output. The same-output partition is the relation

$$E_P = \{(x, y) \mid \boxed{P}(x) = \boxed{P}(y)\}$$

for program P, whose equivalence classes contain inputs leading to identical outputs. Similarly, for a specification S the same-output partition is

$$E_S = \{(x, y) \mid S(x) = S(y)\}.$$

Let z be a particular input for which $\boxed{P}$ is defined. Then z lies in one of the equivalence classes of E_P, in fact the class

$$\{x \mid \boxed{P}(x) = \boxed{P}(z)\}.$$

The case in which the program domain is smaller than the specification domain deserves special comment. Here there exist inputs for which $\boxed{P}$ is not defined, but should be, inputs not in any equivalence class of E_P. Let all such inputs constitute a special "undefined" program class U. It is a primary virtue of specification-based testing that it may select inputs in U.

Intersecting the equivalence classes of E_P (with U added) with those of E_S creates a partition whose classes have members that are literally treated the same in a simple sense. Inputs in one class are all specified to have the same output, and furthermore do have the same output when supplied to the program (or, the program is undefined for all these inputs). For *diagonal* classes the specified and actual output is the same; for *off-diagonal* classes the outputs differ. (U contributes only to off-diagonal classes.)

Theorem: A test using one arbitrary element from each intersection class of the E_P and E_S relations is successful iff P is correct with respect to S.

Proof: (Correctness as a consequence of test success.) Each of the off-diagonal classes must be empty for the test to succeed, because by definition P is wrong for all points therein. Consider then any nonempty diagonal class D, and any $x \in D$. Some $t \in D$ was involved in the successful test, and hence $\boxed{P}(t) = S(t)$. But by definition of D as an intersection class, $S(x) = S(t)$ and $\boxed{P}(x) = \boxed{P}(t)$, hence P is correct, because x was any member of class D. (The reverse implication, test success as a consequence of correctness, is trivial.)

The proof shows that there is an easier way to state this result.

Corollary: The off-diagonal classes of E_P (plus U) and E_S are empty iff P is correct.

Thus in its simplest form, use of homogeneous input classes is a proving technique unconnected with testing: the off-diagonal classes must be shown to be empty, necessarily without testing; there is no need to try points in the diagonal classes.

For example, consider the "triangle problem": triples of integers (A, B, C) representing triangle sides are to be classified into the textbook types such as "scalene." The possible outputs are a finite set, and thus E_S determines a natural finite-index input partition. The natural program that solves the problem has a path corresponding to each possible output, so its path equivalence classes are those of E_P, also of finite index. Fig. 1 shows a possible set of equivalence classes, in which the program fails in two different ways.

Choosing a point from a specification class like "equilateral" may be easy (in Fig. 1: ♡), and a successful test execution shows that the diagonal "equilateral" intersection class is not empty. But it does not prove correctness to proceed in this way, because the tester does not know if it is possible to wander into an off-diagonal class (such as: program prints "equilateral" but the specification requires "isoceles," in Fig. 1: ♦). Indeed, the Corollary states that the off-diagonal classes must be empty for correctness, but it proves nothing to repeatedly fail to select off-diagonal points. The partition is in principle perfect but perfectly useless for testing.

When one class of partition contains inputs *not* treated the same, there is no assurance that success and failure will not be mixed in that class. The partition may then be no help, because the selection problem was not reduced by dividing the input domain. The partition refinement—how much information goes into defining its classes—is of no consequence so long as any large class remains that may contain both success and failure inputs. Thus partitions are flawed in either case: if they are not homogeneous, the testing problem may be no simpler than without them; on the other hand, perfect classes yield a correctness proof, and so in general cannot be obtained. In the one case testing is not improved, in the other it is not involved at all.

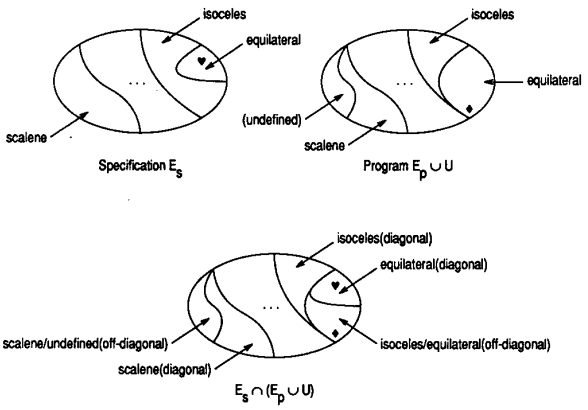


Fig. 1. A possible set of equivalence classes.

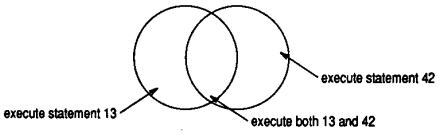


Fig. 2. An example subdivision for statement testing.

B. Overlapping Subdomains

Many kinds of partition testing involve a division of the input domain that is not technically a partition at all, because the subdomains overlap. (Thus the defining relation is not an equivalence relation, because it fails to be transitive.) Two examples are statement testing (defining relation: two inputs are grouped if they cause the same statement to be executed) and mutation testing (grouped inputs kill the same mutant). Such relations do not induce disjoint classes, as shown in Fig. 2.

A true partition can be formed: 1) by intersecting the natural subdomains, removing the intersections from each subdomain, and adding them as separate subdomains (in the figure above, this partition would have three distinct classes with the labels shown); 2) by taking the union of all subdomains with nonempty intersection to be the partition (above, the partition would have one class that includes both subdomains). However, these artificial partitions may be unsatisfactory. For example, in statement testing, 2) always creates a trivial partition with a single equivalence class consisting of the whole input domain, because all inputs cause execution of the first statement in a program. 1) refines the natural subdomains, but the newly created classes do not have the same intuitive meaning as the originals. For statement testing, the subdomains of 1) look rather like path equivalence classes. For other methods, the partitions formed from subdomains may be more natural. In mutation testing, the natural subdomains contain points that kill the same mutant. If the mutation rules are narrow, the subdivisions may fall into many disjoint clusters, so that the partition of 2), whose subdomains contain those tests that kill any one of the mutants in a cluster, seem natural.

It is interesting to examine partition homogeneity when there is overlap. The natural subdomains cannot be homogeneous unless they happen to agree when they intersect. In Fig. 2, for example, all inputs executing 13 cannot lead to failure at the same time that all inputs executing 42 lead to success, for then what of inputs that execute both? If the points of overlap do agree, then homogeneity of the natural subdomains implies homogeneity of both true partitions 1) and 2); this is the only way 2) can be homogeneous. On the other hand, there is nothing to prevent the true partition of type 1) being homogeneous even though the natural subdivisions that defined it are not. In the figure, all inputs executing both 13 and 42 might lead to failure, yet all those executing either statement without the other could lead to success.

The technical results of Section III do not apply to partition testing with overlapping subdomains. However, in Section IV-B the partition 1) is used to analyze an overlapping method.

C. Partition Testing Is Useful

Most of this paper's technical content is critical of partition testing, particularly as a way to inspire confidence in software. However, no one can deny that partition testing is an essential part of software development. The reader should bear in mind the narrow scope of our criticism, which does not impact any of the strengths listed below. (In Section IV-C we suggest improvements to address even the narrow concerns raised in Section III.)

Specification-based testing is valuable early in the development cycle, for example in driving a specification walkthrough. Specification-based subdomains are particularly good at detecting

"missing logic" faults, which are the most common programmer mistake [4]. The systematic nature of partition testing methods recommends them to anyone faced with the real task of generating tests. Furthermore, the generation process can continue as the program develops, incorporating new information as it becomes available.

A subdomain that is nearly homogeneous for *failure* has immense value in debugging. The intersection of specification- and program-based subdomains is valuable because the program part helps to locate the fault, and the specification part indicates what should be done about it. Subdomains based on programmer concerns can have such ideal failure classes. The programmer was worried about (say) collisions in a hash table, and with good reason—when there is a collision the program usually fails.

Catastrophic failures deserve an extra prevention effort, and partition testing is the only way to attack them. These failures may be so infrequent that they are discovered in use only when it is too late, but a partition can be constructed to seek them (for example, using a fault tree).

These virtues of partition testing are double-edged, however. Fault-based partitions put themselves out of business: after all faults of the kind sought have been corrected, there is no reason to believe that the tests which exposed those faults have any further significance. But in confidence testing, the ultimate test is always of this kind: no more failures are found, so the software is released. We will return to this discussion in Section IV, following presentation of our main results.

III. Statistical Analyses of Partitions

Because practical partition testing samples uniformly within classes, it is natural to compare the efficacy of the partitioned test to one in which the same number of points were distributed over the entire input domain. To contrast the latter to "partition testing," we call it "random testing." Although the analysis of partition testing began [3] as a simple comparison of two techniques, we exploit the method to gain a wider goal. Because random testing can be theoretically justified as a confidence measure, we use it to study partition testing. By varying the parameters that relate the two techniques, we can learn what makes partition testing more or less effective. We believe these results to be the first practical testing advice with theoretical backing. We are able to prove what we say, using a theory whose assumptions are completely spelled out. It will take empirical studies to validate the theory—the assumptions may be wrong, after all—but if it proves to be in error, better assumptions can correct it.

The assumptions that we must make, and must examine closely to judge the plausibility of the theory, are tied to the idea of test input distribution. Random testing is viewed as sampling program behavior, so its significance depends on the sample being representative. Partition subdomains, on the other hand, are sampled only to counteract the effects of nonhomogeneity, for which a uniform distribution is appropriate. If there were a natural nonuniform distribution over some subdomain, then further subdivision would be in order to reflect the parts it weights. It is always possible to refine a partition until no further distinctions can be made, and then reflect the input distribution by weighting the sampling rates. The crucial assumption that a theory must make is the extent to which this reflected input distribution distorts the one for random testing. Two extreme assumptions to avoid are the "perfect" partitions that homogeneously isolate failures (because such partitions are unattainable in practice), and at the other extreme, "background" partitions that precisely mimic the random-test input distribution (because they are merely an approximation to random testing). The reader must judge how well we have avoided these extremes. In the situations we investigate, failures are needles in the haystack of the input domain. In hypothesizing how the haystack is partitioned for search, it is difficult to know if we have been fair.

In the discussion to follow presentation of the two statistical theories we will further consider random testing in its own right, but it should be stated at the outset that we do not advocate it as a practical method except in special cases. Rather, we believe it to be the theoretical ideal, and thus perfect for analyzing other methods (here, partition testing).

A. Failure-Rate Model

Duran and Ntafos [3] use the conventional failure-rate model in which a program is assumed to fail at rate θ, and the probability that n independent tests will reveal at least one failure is

$$1 - (1 - \theta)^n.$$

(The tests are assumed to be drawn from the operational distribution.) On the other hand, if n_i points are similarly chosen from class D_i, the class failure rate in each of K classes being θ_i $(1 \le i \le K)$, then the probability of these tests revealing at least one failure is

$$1 - \prod_{i=1}^{K} (1 - \theta_i)^{n_i}.$$

The two formulas immediately above can be compared, and thus the effectiveness of partition testing and random testing can be compared, if the relationship between n, θ, and the n_i, θ_i is established. One should evidently take $\sum_{i=1}^{K} n_i = n$ for a fair comparison: the total number of tests is thus made the same. The relationship between θ and the θ_i is dependent on the input distribution relative to the classes. The remainder of this section describes models of this relationship.

1) Preliminary Investigations: In order to illustrate the method, we will start with the simplest (and least realistic) model. This model assumes that a randomly chosen input is equally likely to fall within each subdivision of the partition, for which

$$\frac{1}{K} \sum_{i=1}^{K} \theta_i = \theta.$$

The difference between random testing and partition testing in terms of the probability of finding at least one failure will be maximum when the variance of the θ_i is at a maximum. For the case where there is one sample per class, this occurs when samples in all the classes except one (say the dth subdomain) have a zero chance of failing, and samples in class d have probability $\theta_d = K\theta = n\theta$ of failing. Table I shows the ratio between the probabilities of finding at least one failure by random testing and by partition testing for a range of θ and K. The data in Table I assume that all of the probability of failure comes from one class. As θ decreases the differences between the methods also decreases. For θ in the range where it should be for released software, the differences are minor unless there are many classes.

TABLE I
RATIO OF PROBABILITIES OF FINDING SOME FAILURE FOR RANDOM VERSUS PARTITION TESTING

Class Count	Failure rate θ				
	0.1	0.01	0.001	0.0001	0.00001
1	1.0	1.0	1.0	1.0	1.0
2	0.95	1.0	1.0	1.0	1.0
5	0.82	0.98	1.0	1.0	1.0
10	0.65	0.96	1.0	1.0	1.0
25		0.89	0.99	1.0	1.0
50		0.79	0.98	1.0	1.0
1000			0.63	0.95	1.0

As long as the probability of a random input being in a class is equal for all classes, the number of classes has to be of the same order of magnitude as the inverse of θ for there to be a significant advantage to partition testing. In addition, there have to be classes with high failure rates. With the most favorable assumptions, the limiting case for this model is that random testing will be $1 - 1/e$ or about 0.63 as effective as partition testing.

2) The Duran and Ntafos Model: Duran and Ntafos [3] investigated a somewhat more realistic model of random-input distribution over the classes. Expressing the probability of detecting an error by random testing in terms of the classes,

$$1 - (1 - \theta)^n = 1 - \left(1 - \sum_{i=1}^{K} p_i \theta_i\right)^n,$$

where p_i is the probability that a randomly selected input will fall in class i. Duran and Ntafos selected the probabilities p_i from a uniform distribution, thus varying the chance that classes are hit by random tests. They then selected θ_i to simulate (nearly) homogeneous classes. Classes were generated with a 2% chance of having a failure rate above 98% and a 98% chance of having a failure rate below 4.9%. Very little data was presented in [3]; only two values for the number of classes (25 and 50) were presented, and n_i was always 1, so that $n = K$. For these parameters, Duran and Ntafos found little difference between partition and random testing.

3) Variations on the Duran and Ntafos Model: These indications are so counterintuitive that we attempted to explore the parameters of the Duran/Ntafos model more fully, but found much the same results as long as we used the same model. We varied the θ_i distribution by changing the balance of classes having high- and low failure rates. We also tried different upper bounds on the low-failure-rate classes and different lower bounds on the high-failure-rate classes. We also tried a wider range of numbers of classes and different numbers of samples per class. The results universally favored partition testing, but not by much. In all cases the random method was at least 80% as effective as the partition method. For example, exploring one case from [3], 25 classes were selected, each having either a "high" failure rate (uniformly distributed from 0.98 to 1.00) or a "low" failure rate (uniformly distributed from 0.049 to 0.00). The probability of a class having a high failure rate was varied. (The second row of the table is the case reported in [3].)

Probability of high failure rate	Probability of finding some failure		
	Partition	Random	Ratio
0.01	0.60	0.54	0.94
0.02	0.75	0.66	0.91
0.04	0.77	0.70	0.93
0.10	0.93	0.85	0.91
0.20	1.0	0.98	0.98
0.30	1.0	1.0	1.0

Thus it does not seem to matter what proportion of the classes are those containing failures.

We next explored the approximation to homogeneous classes. The probability of a class having a high failure rate was held constant at 0.02, the number of classes was 25 and the upper limit of the failure rate in the low failure rate classes was 0.049. The range of failure rates in the high failure rate classes was allowed to vary:

High failure rate Lower Bound	Probability of finding some failure		
	Partition	Random	Ratio
0.80	0.69	0.60	0.90
0.90	0.65	0.58	0.93
0.95	0.63	0.56	0.92
0.99	0.66	0.59	0.93

The homogeneity of the high-failure-rate classes increases toward the bottom of the table, but it does not seem to change the balance between methods.

A similar set of cases were tried for the low-failure-rate upper bound. The probability of a class having a high failure rate was held constant at 0.02, the number of classes was 25 and the lower limit of the failure rate in the high failure rate classes was 0.98. The range of failure rates in the low failure rate classes was allowed to vary:

Low failure rate Upper Bound	Probability of finding some failure		
	Partition	Random	Ratio
0.20	0.96	0.94	0.98
0.10	0.82	0.78	0.96
0.05	0.66	0.60	0.94
0.01	0.56	0.43	0.88

Again, homogeneity increases downward, with only a slight gain for partition testing.

Varying the bounds of both the high- and low-failure-rate classes together produced little difference:

Failure rates bounds		Probability of finding some failure		
High	Low	Partition	Random	Ratio
0.80	0.20	0.96	0.95	0.99
0.90	0.10	0.84	0.81	0.96
0.95	0.05	0.68	0.57	0.88
0.99	0.01	0.49	0.34	0.85

Thus although more homogeneity is better for partition testing, the results are not very sensitive to the approximation used.

Duran and Ntafos note that if random testing is cheaper than partition testing (because the classes need not be devised), then the slight advantage for partition testing could be overcome by using more random tests. In the case they examine, it would require 32 random tests to equal the average effectiveness of 25 classes. Using the above model we were unable to find a situation in which partition testing was clearly superior, and in which varying the parameters exaggerated that superiority. We thus conclude that, in the Duran and Ntafos model, a modest increase in random-testing intensity will always suffice to compensate for partition testing's advantage.

4) Special-Purpose Models: Perhaps partition testing cannot attain a clear advantage because of a deficiency in the Duran/Ntafos model. The model is unrealistic because the overall failure probability is quite high. In reproducing the case described in [3] we found the average failure rate to be $\theta = 0.04$, which is much higher than should be expected in a released program. For the case above with the largest difference between the random and partition methods, the overall failure rate averaged 0.017. In addition, the overall failure rate varied between trials largely because a larger or smaller number of high-failure-rate classes were generated. The largest differences between the testing methods occurred when θ for the individual trials had the highest variability. Another problem with the model is that it assumes that classes are selected purely for their homogeneity and not as a way of focusing test cases in areas where failures are more likely. No correlations were explored between the probability of a random input being in a particular class and the failure rate for that class. The assumption that partitioning methods create a uniform distribution of probabilities of inputs falling in each class also seems unrealistic.

We therefore devised specific models to investigate the questions raised in Section I:

1) Is it important that classes be chosen to find failures (as opposed to seeking coverage of some kind)?
2) How does the efficacy of partition testing depend on class homogeneity?

The first question was addressed by adding to the model above a correlation between the probability of a random input being in a given class and the class's failure rate. The classes with higher failure rates were forced to have a lower probability that a random input would fall in them. In a series of experiments, different weight was given to $1 - \theta_i$ in selecting the class p_i. Aside from the added correlation between p_i and θ_i the parameters were the same as for [3].

Weight of $1 - \theta_i$ used to calculate p_i	Probability of finding some failure		
	Partition	Random	Ratio
0	0.68	0.58	0.90
0.10	0.68	0.58	0.90
0.50	0.69	0.52	0.83
0.90	0.71	0.48	0.77
1.0	0.64	0.47	0.83

At the top of the table, when the weight is 0, the p_i were chosen without considering failure rates, as before. At the bottom of the table, a weight of 1.0 means that the choice of p_i is no longer random, but completely determined by $1 - \theta_i$. As expected, forcing the "random" inputs to concentrate in the classes where failure is less likely (at the end of the table) did hurt the effectiveness of random testing, but not dramatically. Since the overall failure rate for the model was still high we decided to try a different model where the failure rate could be controlled.

The new model assumed that a partitioning method created two types of classes. One type ("hidden classes") had a small probability of being hit by a random input (from 0.001 to 10^{-8}); the other type ("exposed classes") had a high probability of being hit (from 0.04 to nearly 1.00). In other words, the hidden classes were very unlikely to be selected during random testing, while always being selected during partition testing. In addition, we assumed low overall failure rates (from 0.001 to 10^{-8}). For a given set of probabilities of a random input being in a given class and a given overall failure rate we varied the failure rate for the hidden classes from 1 to 0. The failure rate in the exposed classes was adjusted to maintain a constant overall rate and, therefore, a constant chance of finding a failure by random testing. We then calculated the probabilities of finding at least one failure using the partition method. As would be expected, the partition method was clearly superior to the random method when the hidden classes had failure rates higher than the overall failure rate. When the hidden classes had a lower rate than the overall rate the random method was superior. For example, for an overall failure rate of 0.00001 and 25 classes, the random method will find a failure with probability 0.00025. If 20 of the classes are hidden the partition method behaves as follows:

Hidden class failure rate	Probability of finding some failure	
1.0	1.0	
0.1	0.88	
0.01	0.18	
0.001	0.020	
0.0001	0.0020	
0.00001	0.00025	← (break even)
0.000001	0.000070	
0.0000001	0.000052	
0	0.000050	

When the failure rate in the hidden classes was greater than the overall rate, increasing the number of hidden classes increased the advantages of partition testing. However, when the failure

rate was lower in the hidden classes, increasing the number of hidden classes caused the partition method to fare worse.

In another experiment, failure rates were randomly generated for 24 hidden classes. The failure rate in a single exposed class was adjusted to maintain a constant overall failure rate. In 100 trials the failure rate was set to 1 at random for a low proportion of the hidden classes. For low overall failure rates the partition method was clearly superior to the random method when the probability of a failure rate of 1 was 0.01 or greater in the hidden classes. If the probability of a failure rate of 1 was reduced to 0.001 the random method was significantly better.

The second question to be addressed by the special purpose models concerned the importance of homogeneity. When classes are precisely homogeneous, the assumption that a point is selected from each class amounts to proving correctness, and in that case partition testing should outperform random testing by any desired factor, the more so the smaller the failure probability. (However, as pointed out in Section II-A, the case of perfect homogeneity has no practical significance, because points cannot necessarily be found in "failure" or off-diagonal classes.) We expected to observe the effects of homogeneity in the above simulations, but did not, so a special experiment was performed. In the last model described above, high failure rates in the hidden classes were permitted to vary uniformly from 1 to 0.2 (low homogeneity) and the results were compared to the case where they varied from 1 to only 0.9 (high homogeneity). The largest impact of low homogeneity was found to be only a 22% decrease in the effectiveness of partition testing. In cases where very few of the classes (0.001 or less) contain a high proportion of failures, class heterogeneity has no effect.

5) Failure-Rate Model Summary: These special-model results indicate that the advantage of partition testing arises only from increased sampling in regions where failures can occur. In other words, if you already know where failures are likely, this information can usefully partition the input space. This is contrary to the discussion in [3] where the homogeneity of the classes is the important factor. If this result is correct, it calls into question the basis of most testing methods based on coverage, which partition on the basis of assumed homogeneity and ignore failure probability. Finding a few classes with failure rates much greater than the overall rate is more useful than finding many nearly homogeneous classes with undistinguished failure rates.

B. Defect-Rate Model

The fundamental assumptions of the failure-rate model are called into question in [5], and a probable-correctness model proposed instead. It is argued that faults are uniformly distributed in the state space of the program code, not over the input space of program executions. The valid partitions are therefore those that result from reflecting uniform coverage of program states into the input domain where testing must be done. Of course, sampling according to this model is even less practical than trying to observe an unknown operational distribution. The reflected distribution cannot be calculated even in principle, because it is an unsolvable problem to determine the data states that can reach an arbitrary control point. However, using an idea of Valiant's [10], it is still possible to relate the number of test points to the probability of missing a failure. We cannot in general find an ideal partition, but we can compare methods on the assumption that an ideal partition was used.

1) Valiant's Model: Valiant's statistical model describes the situation in which independent drawings are made from a space containing many kinds of objects. When drawings fail to turn up any new kinds, how likely is it that they are still present? The application to testing for confidence is that the space is all program executions, the drawings are tests, and failures are never drawn. How many successful trials N are needed to have confidence that the probability of unseen failures in the space is less than p? A relationship between N, p, and the size of the space K is derived in [5] (and reexamined below), from which N can be calculated. This result can be used to compare partition testing to random testing.

Throughout this paper we have used "confidence" in its nontechnical sense expressing the belief that a program is of good quality, unlikely to fail. In this section the word will instead have its technical meaning of a probability that another probability is correct. We return to the nontechnical meaning in Section IV.

The learning-theory application for which Valiant devised his model is only concerned with the probability of unseen objects in a fixed space. The calculation is not exact, and its quality depends on the size of the space K. Thus in comparing two different spaces the results may be invalid if the inaccuracies lie in the wrong direction. We repeat the derivation with attention to these details and the application to testing; it turns out that both [10] and [5] are incorrect (although the qualitative results do not change). We also separate the confidence probability from the probability that failures have been missed, which neither [10] nor [5] investigated.

Suppose a space of K kinds of successful program executions is sampled, with the result that no failures are observed. We wish to establish that the fraction of failures that might exist unseen in the space is less than p, and to do so with a confidence e, by selecting a sample of N points without a failure. Valiant's idea was to count the number of ways the samples could be dispersed over the different kinds of executions. A dispersion of s, i.e., that exactly $s + 1$ different kinds were chosen among the N points, can be viewed as exposing s differences in the drawing. For example, $s = 0$ means that there were no differences exposed, because $s + 1 = 1$ kinds were drawn. Suppose that the probability of drawing a different kind were d, independent of the number of points drawn. Then the probability of dispersion s is given by the Bernoulli distribution:

$$\binom{N}{s} d^s (1-d)^{N-s}.$$

Dispersions from 0 to $K - 1$ are independently possible without drawing a failure, so the probability that N points will all be successful executions is

$$\sum_{s=0}^{K-1} \binom{N}{s} d^s (1-d)^{N-s}.$$

Valiant argues that although the probability of drawing a different kind is not constant—indeed it decreases as more kinds have been drawn—it is always greater than p, the probability of drawing a failure, because all failures remain undrawn. Because the summation above is nondecreasing when d decreases [11], replacing d with the smaller p overestimates the probability, so that N is an upper bound, establishing confidence e that the failures occupy no more than fraction p of the space when

$$\sum_{s=0}^{K-1} \binom{N}{s} p^s (1-p)^{N-s} < 1 - e.$$

(In [10] the size of the space is wrong, e.g., $K = 2$ for {success, failure} instead of $K = 1$; in [5] the summation has the right number of terms, but starts at 1 instead of 0.)

It is argued in [5] that failure fraction p must be very small to obtain confidence in software being correct; however, it is less clear that the confidence e needs to be extreme. Some typical values for $p = 10^{-6}$ are:

K	e (%)	N ($\times 10^6$)
1	99	5
1	99.999	12
10	99	19
10	99.999	29
100	99	120
100	99.999	150

In contrast to the behavior as K increases, improving e does not substantially increase the samples required, and the sensitivity decreases with increasing K. Similar behavior is observed as p varies over six orders of magnitude: N does not vary greatly with e. To a crude approximation, for larger K and smaller p, N increases directly, but N changes slowly with e.

2) Testing Comparison: Valiant's model can be used to compare partition testing to overall random testing. The comparison is quite different from the ones presented in Section III-A, because it is in terms of unseen failures. Nothing is assumed about the probability these will ever come to light in using the software, merely that they exist. Thus we are looking for all possible defects, a form of assurance testing that is far more stringent than mere operational reliability. Consider testing in L subdomains of a partition, seeking to establish a probability h that no failures are possible. Assuming the subdomains cannot be refined, the execution space for each is only {success, failure}, i.e., $K = 1$. Let the probability that a failure exists in each subdomain be p. Then the probability that there is no failure in each is $1 - p$, and that there is no failure in them all is $(1 - p)^L$. Then the probability of at least one hidden failure is $h = 1 - (1 - p)^L$. Solving for p, we obtain the probability that must be attained in the subdomains to gain the overall h (which is of course more stringent than h): $p = 1 - (1 - h)^{1/L}$. Thus the number of samples needed in each subdomain is N_p, which can be calculated from the formula above using this p and $K = 1$.

On the other hand, overall random sampling requires N_R samples, in the formula using h directly, in a space of no more than $K = 2L$ kinds of objects (success/failure in each subdomain). The ratio LN_P/N_R is a measure of the quality of the two testing methods, since the respective number of points give the same probability for undiscovered failures. The result in [5] is that partition testing requires many times more tests, and the disparity *widens* with more subdomains and more stringent probabilities. The model presented in Section III-A suggests that partition testing will only outperform random testing for a small number of subdomains, and high failure probability, cases not investigated in [5]. Taking $L = 20$ subdomains, the ratio of points required by partition testing to those required for random testing at the same confidence (e) and failure probability ($h = 0.001$) was investigated as follows:

e (%)	N_R	LN_P/N_R
99.999	62 400	89
90	48 300	38
50	39 700	14
5	30 200	2.4
1	26 800	0.30

Thus random testing outperforms partition testing by many times, unless very low confidence is to be placed in the results. Similar results are obtained for wide variations in number of subdomains (partition performance is better for fewer subdomains) and failure probabilities (partition performance declines as failure is less likely).

3) Discussion of Valiant's Model: When $K = 1$ the formula above is exact, because only dispersion 0 is possible, and the chance of drawing N executions of the same kind is $(1 - p)^N$, in agreement with conventional decision theory. But for $K > 1$ the calculated bound N is too large, because the probability p is an underestimate. In the comparison of partition and random testing, the choice of $K = 2L$ for random testing also leads to overestimating N_R, because the space might be as small as L (if none of the subdomains contained failures). The results therefore make random testing appear worse than it is. Thus the balance in favor of random testing is increased rather than mitigated by the inaccuracies in the calculation.

Partition testing should be a "divide and conquer" strategy, but under the assumptions of the defect-rate theory it is really a "divide and founder" one. L subdomains turn one problem into L problems, and since a given level of assurance in the whole requires a higher assurance level in each part, each new problem is *more* difficult than the original. (The small subdomains of the partition decrease N_P, but not so much as the more stringent assurance increases it.) This observation explains the results, but does not help much in explaining their counterintuitive nature. The difficulty may lie with the model itself. Subdividing test success/failures by attaching the subdomain from which the test input came, is not clearly an accurate description of the overall random test situation. The subdivision seems arbitrary, unconnected to the test method, yet it influences the result. The assumption that tests are independent samples from the space of all executions is also suspect: how can this be arranged through the input domain that favors some execution patterns and makes others extremely difficult to excite?

IV. Discussion: How Should We Test?

It should be a goal of testing theory to provide sound advice for those who practice the testing art. In this section we provide practical advice based on the theory of Section III.

A. Confidence Versus Failure

Our main point has been to call into question the common wisdom that confidence in software is obtained by vigorously seeking failures, and when a variety of (partition-testing) methods finds no more failures, concluding that the software will prove reliable in use. Unless the methods used employ orders of magnitude more test points than is usual, this conclusion is false. Random testing is the only standard for reliability in use,

and we have shown in Section III-A that partition testing has a hard time improving on it. About 4.6 million successful inputs drawn from the operational distribution are required to attain 99% confidence that a program will fail less than one time in 10^{-6}. In partition testing which is about as good as random testing, a (say) specification-based partition test in which one test is tried in each of 100 subdomains, gives high confidence only that the chance of failure is less than about 1 in 25. Clever choice of a partition in no way compensates for the disparity in these numbers.

A second, similar point is that the situation is even worse if the tested software is to be assessed for defects. Further orders of magnitude more tests are required, so many that these will never be practical; and, partitions perform badly. The theory of Section III-B is more questionable than that for conventional reliability, but its results are provocative.

B. Application to Overlapping Subdomains

The "natural" subdomains of partition methods are those that arise from the method's systematic character. For example, a person seeking to attain statement coverage considers one statement at a time, and hence the natural subdomains are test inputs that execute each statement. Similarly, a mutation tester tries to kill individual mutants, so inputs that kill a mutant are a natural subdomain. As Section II-B notes, the natural subdomains do not form a partition for these and some other methods, so the results of Section III do not apply directly. A true partition can always be formed from a collection of overlapping subdomains by treating intersections as distinct classes. In this section we apply the results of Section III to a partition obtained from an overlapping method, to analyze the original method.

Much of Section III-A was devoted to investigating the way an overall random test on a domain falls among the subdomains that partition that domain. The results can now be applied in miniature. Consider a natural subdomain as the whole, and its intersections with other natural domains as the partition classes. The rough results of Section III-A are that it will be difficult to see the difference between "partition testing" (that is, choices in the artificial, true partition), and "random testing" (that is, choices from the original subdomain). It is not quite that simple, however, because each natural subdomain is not isolated. Choices made in two natural subdomains may both fall in the intersection, so that in the true partition, some classes are sampled more heavily than others. Again, results of Section III-A apply. If the heavily sampled intersection classes have a high failure rate, the original test, considered on the artificial partition, will appear better; or, if the intersection classes have about the same failure rates as the natural subdomains of which they are part (or lower failure rates), the true partition test will be the same (or less effective), compared to the original.

Detailed analysis of particular methods will be required to discover if their overlap is beneficial or harmful. In general, we can say that benefits are likely to be short-lived in the testing process. Suppose an overlapping part of several subdomains has a high failure rate, and so partition testing in the overlapping method is really better than it seems. Once the faults that were exposed are fixed, the overlap class loses its usefulness.

The results of Section III-B seem to apply to methods with overlapping subdomains, but partition testing may do even more poorly. The true partition formed by intersecting the natural subdomains has new classes, and the number of samples is divided, perhaps increasing in overlap classes, but certainly decreasing where there is no overlap. Without attempting a careful analysis, it appears that the negative factors will outweigh the positive.

In any case, nothing in overlapping subdomains mitigates the disparity between the number of test points required for confidence and the number usually used in partition testing. It could happen that overlap increased the sampling rate in an intersection class manyfold, and that class might also be heavily weighed in the operational distribution. But a difference of several orders of magnitude is difficult to make up.

C. Improved Partition Testing

It is not unfair to say that partition-testing methods have been devised to meet the twin criteria of systematic coverage of software features and homogeneous subdomains. Covering something that is manifest in programs allows the testing to be systematic; homogeneity is the ideal circumstance under which partitions should work. Our results of Section III-A call both criteria into question. Undifferentiated coverage is the antithesis of concentrating failures in subdomains; in the imperfect practical case homogeneity is not very important. Thus existing partition testing methods, created to meet irrelevant criteria, may admit considerable improvement. (Failure-finding ability is the issue addressed by our theory; of course, systematic coverage is still useful in devising tests, and we do not suggest that it be abandoned.)

Careful analysis of existing methods is needed to see how well they create subdomains with high failure rates, and to be sure about the consequences of subdomain overlap. We expect to see some quantitative differences between existing methods, backing up the intuition that some methods (notably mutation) seem to be "harder to fool" than others (notably path testing). It should be especially interesting to take a hard look at methods that seem to be partitions just for the sake of division, for example specification-based blackbox testing, or intersection of subdomains arising from different methods. The theory predicts that these are the least valuable for exposing failures, yet they are among the most trusted in practice.

We apply the theory to suggest a new partition-testing method that should expose failures, because it creates subdomains with high failure rates. Let us call this method "suspicion testing." Its subdomains (which are overlapping) are defined by identifying inputs that probe parts of the software likely to fail. The sources of failure are obvious and well known; yet, suspicion testing is not commonly practiced. The following identify routines in a developing system that may be expected to cause trouble:

1) A routine written by the least experienced programmer on the project.
2) A routine with a history of failure, either in development or in the field.
3) A routine that failed an inspection and had to be substantially modified late in the design cycle.
4) A routine involved in a late design change, begun after most coding was complete.
5) A routine about which a designer or coder feels uneasy.

The reader can no doubt add to this list.

At the system level, suspicion testing defines a kind of gross "routine coverage"—tests must invoke each routine—weighted to emphasize troublesome routines. At the unit level, suspicion testing requires that testing methods used on troublesome routines be more extensive than usual. For example, branch coverage could require many distinct tests to take each alternative, or that two independent branch-coverage test sets be devised, or that if

N random tests are needed to attain branch coverage, that $5N$ be used instead.

V. Summary and Suggestions for Future Work

It has been shown that partition testing is not a good technique for inspiring confidence in a program through successful tests. A combination of theoretical and statistical analysis indicates that its success occurs only when subdomains with a high failure probability can be identified—that is, when the failures are suspected and localized. This is not a bad thing, and such subdomains are well suited to debugging. But the usual release test based on partition testing, that is, one in which no failures are observed, is no better than a random test without subdomains. And because the test points selected in each subdomain are few and the selections are not independent, as a random test it has very small significance. If the release test is constructed from debugging tests, it is particularly untrustworthy, since the small, failure-prone subdomains that are good for debugging are just the wrong ones to inspire confidence.

If random testing is to be a viable alternative to partition testing, there must be a shift from people-intensive test design to machine-intensive automatic test generation. Testing is a problem that parallelizes perfectly, so the concurrent processing power that is rapidly coming into use may supply the machine resources economically. But random testing with orders of magnitude more tests than are used today is impractical without a test oracle to mechanically judge the results. In unusual situations there is a simulation program, an old software system that is being replaced, or even a physical system that can be measured to serve as an oracle. Two special cases deserve mention. In one, testing a protocol or a coding/decoding scheme, the output should be the same as the input, which is easy to check mechanically. In another, testing a Postscript interpreter for a color printer, random-test output was visually compared to samples from similar printers [12]. In the typical case, however, no oracle exists. Research on formal, effective specifications is therefore of the first importance for testing.

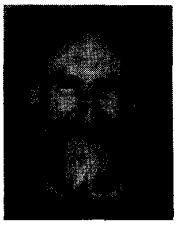
All partition-testing methods are suspect when used to gain confidence in software. Particularly suspect are the important cases of quality assurance in which haphazard test collections are augmented to achieve coverage (but no failures appear), and the case of regression testing in which coverage is attained without failures. Our theory suggests that to demonstrate the efficacy of such methods even for debugging, their proponents must show a connection between the method's subdomains and likely failures, and must study the question of overlap.

Acknowledgment

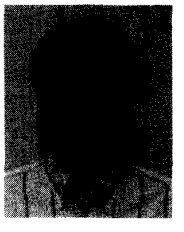
E. Weyuker noticed that subdomains overlap and are thus not equivalence classes.

References

[1] J. Goodenough and S. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.,* vol. SE-1, pp. 156–173, 1975.
[2] D. Richardson and L. Clarke, "A partition analysis method to increase program reliability," in *Proc. 5th Int. Conf. Software Engineering,* San Diego, CA, 1981, pp. 244–253.
[3] J. Duran and S. Ntafos, "An evaluation of random testing," *IEEE Trans. Software Eng.,* vol. SE-10, pp. 438–444, July 1984.
[4] R. Glass, *Modern Programming Practices.* Englewood Cliffs, NJ: Prentice-Hall, 1982.
[5] R. Hamlet, "Probable correctness theory," *Inform. Processing Lett.,* vol. 25, pp. 17–25, Apr. 1987.
[6] E. Weyuker and T. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Trans. Software Eng.,* vol. SE-6, pp. 236–246, 1980.
[7] L. Morell, "A model for assessing code-based testing techniques," in *Proc. 5th Annu. Pacific Northwest Software Quality Conf.,* Portland, OR, 1987, pp. 309–325.
[8] H. Mills, V. Basili, J. Gannon, and R. Hamlet, *Principles of Computer Programming: A Mathematical Approach.* Boston, MA, Allyn and Bacon, 1987.
[9] G. Myers, *The Art of Software Testing.* New York: Wiley, 1979.
[10] L. G. Valiant, "A theory of the learnable," *Commun. ACM,* vol. 27, pp. 1134–1142, Nov. 1984.
[11] D. Angluin and P. Laird, "Identifying k-CNF formulas from noisy examples," Yale Univ., Tech. Rep. YALEU/DCS/TR-478, June 1986.
[12] R. Taylor, "An example of large-scale random testing," in *Proc. 7th Annu. Pacific Northwest Software Quality Conf.,* Portland, OR, 1989, pp. 339–348.



Dick Hamlet (M'81) is a Professor of Computer Science at Portland State University, Portland, OR. After a career in electrical engineering, engineering physics, and mathematics, he discovered computing, in the form of recursion theory, at the University of Washington, Seattle. He now concentrates on software engineering theory, especially testing theory.



Ross Taylor received the A.B. degree in economics and the M.B.A. degree in accounting from the University of California in 1975 and 1980, respectively.

He has worked as a management information consultant and a financial systems analyst and is certified in Production and Inventory Control at the Fellow level. Presently he is a software engineer in the Graphics Printing and Imaging Division of Tektronix, Inc., working in the area of software quality assurance.